

10-millisecond Computing

Gang Lu*, Jianfeng Zhan^{*,†}, Tianshu Hao*, and Lei Wang^{*,‡}

^{*}Beijing Academy of Frontier Science and Technology

^{*}Institute of Computing Technology, Chinese Academy of Sciences

[‡]University of Chinese Academy of Sciences

lugang@mail.bafst.com, zhanjianfeng@ict.ac.cn, haotianshu@ict.ac.cn,
wanglei_2011@ict.ac.cn

Abstract

Despite computation becomes much complex on data with unprecedented large-scale, we argue computers or smart devices should and will consistently provide information and knowledge to human being in the order of a few tens milliseconds. We coin a new term 10-millisecond computing to call attention to this class of workloads.

Public reports indicate that internet service users are sensitive to the service or job-level response time outliers, so we propose a very simple but powerful metric—outlier proportion to characterize the system behaviors. The outlier proportion is defined as follows: for N completed requests or jobs, if M jobs or requests' latencies exceed the outlier limit t , e.g. 10 milliseconds, the outlier proportion is $\frac{M}{N}$.

10-millisecond computing raises many challenges for both software and hardware stacks. In this paper, as a case study we investigate the challenges raised for conventional operating systems. For typical latency-critical services running with Linux on a 40-core server—a main-stream server hardware system in near future, we found, when the outlier limit decreases, the outlier proportion of a single server will significantly deteriorate. Meanwhile, the outlier proportion is further amplified by the system scale, including the system core number. For a 1K-scale system, we surprisingly find that to reduce the service or job-level outlier proportion to 10%, running Linux (version 2.6.32) or LXC (version 0.7.5) or XEN (version 4.0.0), respectively, the outlier proportion of a single server needs to be reduced by 871X, 2372X, 2372X accordingly. We also conducted a list of experiments to reveal the current Linux systems still suffer from poor outlier performance, including Linux kernel version 3.17.4, Linux kernel version 2.6.35M, a modified version of 2.6.35 integrated with sloppy counters proposed by Boyd-Wickizer et al. and two representative real time schedulers.

1. Introduction

Despite computation becomes much complex on data with unprecedented scale, in this paper we argue computers or smart devices should and will consistently provide information and knowledge to human being in the order of a few tens milliseconds. We coin a new term 10-millisecond computing to call attention to this class of workloads.

First, determined by the nature of human being's nervous and motor systems, the timescale for many human activities is in the order of a few hundreds milliseconds [14, 7, 8]. For example, in a talk, the gaps we leave in speech to tell the other person it is 'your turn' are only a few hundred milliseconds long [14]; the response time of our visual system to a very brief pulse of light and its duration is also in this order. Second, the experiments in [7] show perceptual events occurring within a single cycle (of this timescale) are combined into a single percept if they are sufficiently similar, indicating our perceptual system cannot provide much *finer* capability. Third, one of the key results from early work on delays in command line interfaces is that regularity is of the vital importance [14, 7, 8]. If people can predict how long they are likely to wait they are far happier [14, 7, 8]. So perfect human-computer interactions come from human being's requirements, and should be irrelevant to the scale of data, complexity of tasks, and underlying hardware and software systems. Finally, the above observations have been confirmed by internet services users behaviours. In fact, they will not lower their QoS expectation because of the complexity of underlying infrastructures. Instead, keeping low latency low is of the vital importance for attracting and retaining users [12, 8, 21].

Google [27] and Amazon [21] found that moving from a 10-result page loading in 0.4 seconds to a 30-result page loading in 0.9 seconds caused a decrease of 20% of the traffic and revenue; Moreover delaying the page in increments of 100 milliseconds would result in substantial and costly drops in revenue. As internet users are sensitive to the service or job-level response time outliers, we propose a very simple metric—outlier proportion to characterize the system behaviors. For N completed requests or jobs, if M jobs or requests' latencies exceed the outlier limit t , e.g. 10 milliseconds, the outlier proportion is $\frac{M}{N}$. Section 2 discusses several advantages of the outlier proportion metric with respect to the $n\%$ tail latency metric that is the mean latency of all requests beyond a certain percentile $n\%$.

10-millisecond computing raises many challenges for both software and hardware stack. In this paper, as a case study we investigate the challenges raised for conventional operating systems. For a typical latency-critical services, mem-

cached, running with Linux, on a 40-core server, we found, when the outlier limit decreases, the outlier proportion of a single server will significantly deteriorate. Meanwhile, the outlier proportion also deteriorates as the system core number increases. The outlier is further amplified by the system scale. For a 1K-scale system—a typical configuration in internet services, we surprisingly find that to reduce the outlier proportion of the service or job to 10%, running Linux (version 2.6.32) or LXC (version 0.7.5) or XEN (version 4.0.0), the outlier proportion of a single server needs to be reduced by 871X, 2372X, 2372X accordingly. We also conducted a list of experiments to reveal the current Linux systems still suffer from poor outlier performance. The *operating systems* we tested are Linux with different kernels: 1) 2.6.32, an old kernel released five years ago but still popularly used and in long-term maintenance. 2) 3.17.4, a latest kernel released on November 21, 2014. 3) 2.6.35M, a modified version of 2.6.35 integrated with *sloppy counters* proposed by Boyd-Wickizer et al. to solve scalability problem and mitigate kernel contentions [6] [2]. 4) representative *real time schedulers*, SCHED_FIFO (First In First Out) and SCHED_RR (Round Robin). This observation indicates the new challenges are significantly different from traditional outlier and stagger issues widely investigated in MapReduce and other environments [26, 18, 22, 23, 29].

We also note that the devices interacting with users become much light-weighted, and power-constrained, and hence complex computation must be offloaded to back-end datacenters [10] to enable perfect computer-user interaction. The recent work [28] argues for breaking data-parallel jobs in compute clusters into tiny tasks that each complete in hundreds of milliseconds for batch and interactive sharing and straggler mitigation. We believe more 10-milliseconds workloads will emerge, which will further boom back-end datacenters.

2. Problem Statement

For scale-out architecture, a feasible solution is to break data-parallel jobs in computer clusters into tiny tasks [28]. On the other hand, for large-scale online services, a request is often fanned out from a root server to a large number of leaf servers (handling sub-requests) and responses are merged via a request-distribution tree [12].

We can use a probability function $Pr(T \leq t)$ where $T \geq 0$ to describe the distribution of service or job-level response time (T). If SC ($SC \geq 0$) leaf servers (or slave nodes) are used to handle sub-requests or tasks sent from the root server (or master node), we use T_i to denote the response time of a task or sub-request on server i . Here, for clarity, we intentionally ignore the overhead of merging responses from different sub-requests. Meanwhile, for the case of breaking a large job into tiny tasks, we only consider the most simplest scenario—one-round tasks are merged into results, excluding the iterative computation scenarios.

The service or job-level outlier proportion is defined as fol-

lows: for N completed requests or jobs, if M jobs or requests' latencies exceed the *outlier limit* t , e.g. 10 milliseconds, the outlier proportion $op_sj(t)$ is $\frac{M}{N}$.

According to [12], the service or job-level outlier proportion will be extraordinarily magnified by the system scale SC .

The outlier proportion of a single server can be represented by $op(t) = Pr(T > t) = 1 - Pr(T \leq t)$.

Assuming the servers are independent from each other, the service or job-level outlier proportion, $op_sj(t)$, can be denoted by Equation 1

$$op_sj(t) = Pr(T_1 \geq t \text{ or } T_2 \geq t, \dots, \text{ or } T_{SC} \geq t) \quad (1)$$

$$= 1 - Pr(T_1 \leq t)Pr(T_2 \leq t) \dots Pr(T_{SC} \leq t) \quad (2)$$

$$= 1 - Pr(T \leq t)^{SC} = 1 - (1 - Pr(T > t))^{SC} \quad (3)$$

$$= 1 - (1 - op(t))^{SC} \quad (4)$$

When we deploy the XEN or LXC/Docker solutions, the service or job-level response outlier proportion will be further amplified by the number K of guest Oses or containers deployed on each server.

$$op_rs(t) = Pr(T_1 \geq t \text{ or } T_2 \geq t, \dots, \text{ or } T_{SC \times K} \geq t) \quad (5)$$

$$= 1 - (1 - op(t))^{SC \times K} \quad (6)$$

On the vice versa, to reduce an service or job-level outlier proportion to be $op_sj(t)$, the outlier proportion of a single server must be low as shown in Equation 7.

$$op(t) = 1 - \sqrt[SC]{1 - op_sj(t)} \quad (7)$$

For example, a Bing search may access 10,000 index servers [25]. If we need to reduce the service or job-level outlier proportion to be less than 10%, the outlier proportion of a single server must be low as $1 - (0.9)^{1/10000} \approx 0.000011$. Unfortunately, Section 3 shows it is an impossible task for the conventional OS like Linux to provide such capability.

From a cost-effective perspective, another important performance goal is the valid throughput—how many requests or jobs can be finished within the outlier limit. In fact, according to the outlier proportion, it is very easy to derive the valid throughput. According to the definition of the outlier proportion, N is the throughput number. The valid throughput is $(N - M)$.

Another widely-used metrics is *n% tail latency*. For the completeness, we also include its definition. The *n%* tail latency is the mean latency of all requests beyond a certain percentile n [20], e.g., the 99th percentile latency. Outlier proportion and tail latency are two related concepts, however, there are subtle differences between two concepts. With respect to the metric *n%* tail latency, the outlier proportion is much easier to be used to represent both the user requirement,

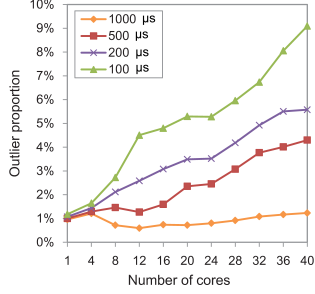


Figure 1: The outlier proportion of memcached on each server changes with the outlier limits (from 100 to 1000 μ s) and core numbers (from 1 to 40).

e.g., $(\frac{N-M}{N})$ requests or jobs satisfying with the outlier limit and the service provider’s concern, e.g., the valid throughput $(N - M)$ measuring how many requests or jobs finished within the outlier limit. We also note that we cannot derive the valid throughput from the $n\%$ tail latency. As the outlier proportion does not rely upon the history latency data, while the tail latency needs to calculate the average latency of all requests beyond a certain percentile, so the former will be much robust to latency data variability and easier to handle in the QoS guarantee. Moreover, as shown in Equation 7, the outlier proportion of each server is easily derived from the service or job-level outlier proportion when the system scale is known. However, it is very difficult to establish the relationship between tail latency among a single server and the whole service or job.

2.1. Related concepts

Soft real time systems are those in which timing requirements are statistically defined [5]. An example can be a video conference system it is desirable that frames are not skipped but it is acceptable if a frame or two is occasionally missed. The goal of a soft real time system is not to reduce the service or job-level outlier proportion but to reduce the average latency within a specified threshold. If we use the tail latency to describe, that is 0% tail latency must be less than the threshold. Instead, in a hard real time system, the deadlines must be guaranteed. That is to say, the service or job-level outlier proportion must be zero! For example if during a rocket engine test this engine begins to overheat the shutdown procedure must be completed in time [5].

3. Challenges from an OS perspective

We investigate the outlier problem from a perspectives of the operating system using a latency-critical workload: memcached. memcached [1] is a popular in-memory key-value store intended for speeding up dynamic web applications by alleviating database loads. The average latency is about tens or hundreds μ s. A real-world memcached-based application usually need to invoke several *get* or *put* memcached operations, in addition to many other procedures, to serve a single request, so we choose it as a case study on 10-millisecond

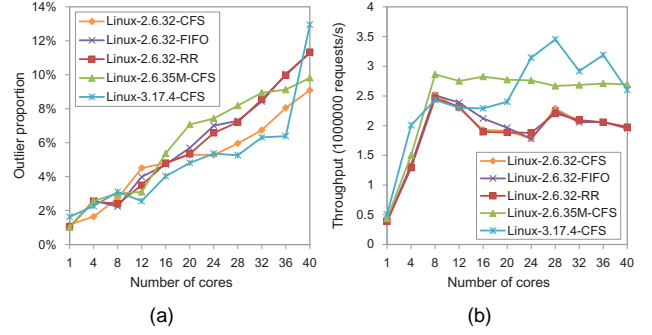


Figure 2: (a) shows the outlier proportions of a single server using different Linux kernels and schedulers. (b) shows the valid throughput. The core number of a server is varied (see X axis). The outlier limit is 100 μ s.

computing. We increase the running cores and bind a memcached instance on each active core via *numactl*. A 40-core ccNUMA (cache coherent Non Uniform Memory Access) machine with four NUMA domains (Each with a 10-core E7-8870 processor.) is used for running memcached instances. Four 12-core servers running client emulators which are obtained from MOSBench [6]. The host OS is SUSE11SP1 with kernel version 2.6.32 and a default scheduler CFS (Completely Fair Scheduler).

Figure 1 shows the outlier proportions under different outlier limits and increasing cores. We can observe that: a) tighter outlier limits lead to higher outlier proportions. We can get a low outlier proportion of 0.60% on a common 12-core node with the outlier limit of 1 second. By contrast, it will be high as 4.50% if we reduce the outlier time limit to 100- μ s. b) The outlier proportion increases gradually with the number of cores. In the worst cases, it degrades to 9.09%. According to Equation 1, using 1K 40-core servers, the service or job-level response time outlier proportion will be up to nearly 100%.

Following the above observations, we now discuss the challenges for a monolithic kernel (Linux) and virtualization technologies including Xen and Linux Containers.

3.1. A monolithic kernel: Linux

We conducted a list of experiments to study whether current Linux systems still suffer from poor outlier performance. The *operating systems* we tested are Linux with different kernels: 1) 2.6.32, an old kernel released five years ago but still popularly used and in long-term maintenance. 2) 3.17.4, a latest kernel released on November 21, 2014. 3) 2.6.35M, a modified version of 2.6.35 integrated with *sloppy counters* proposed by Boyd-Wickizer et al. to solve scalability problem and mitigate kernel contentions [6] [2]. *sloppy counters* adopts local counters on each core and a central counter to avoid unnecessary touching of the global reference count. Their evaluations show its good effects on mitigating unnecessary contentions on kernel objects. Beside these systems

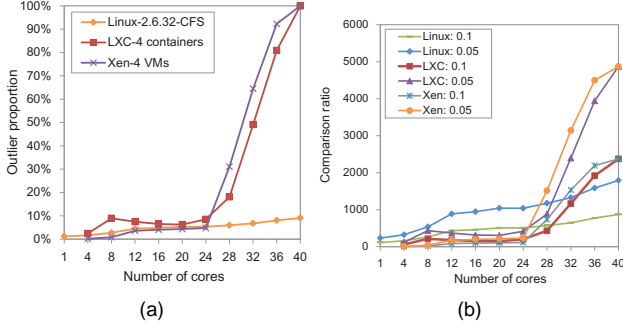


Figure 3: (a) shows the outlier proportions of a single server when running Linux, 4 containers (LXC) and 4 VMs (XEN) on the varied core number (see X axis). The outlier limit is 100 μ s. (b) shows for 1K-server configuration, how many times of the outlier proportion (see Y axis) of each server need to be reduced to guarantee the service or job-level response outlier proportion is 10% or 5%. The core number of a server is varied (see X axis). The outlier limit is 100 μ s.

with different kernels, we also evaluated the impact of representative *real time schedulers*, SCHED_FIFO (First In First Out) and SCHED_RR (Round Robin). A SCHED_FIFO process runs until either it is blocked by an I/O request (if a higher priority process is ready) or itself invokes *sched_yield*. SCHED_RR is a simple implementation of SCHED_FIFO, except that each is allowed to run for a maximum time slice. The time slice in SCHED_RR is set to 100 μ s. We use a dash with CFS, FIFO, or RR following the kernel version to denote the scheduler the kernel uses in the figures.

Results and discussions. Figure 2a and Figure 2b show that these systems are not competent for low outlier proportion under the time limit of 100 μ s. First, all systems has a scalability problem on the valid throughput. Although the modified kernel Linux-2.6.35M behaves better than Linux-2.6.32, the throughput stops increasing after 8 cores. Second, the outlier proportions climb up to 10% with 40 cores. Besides, the latest kernel 3.17.4 still cannot surpass the older kernel 2.6.35M. Such a ad-hoc method of mitigating potential resource contentions seems to be endless and contribute to limited improvements. Third, real time schedulers neither reduce outlier proportion nor improve throughput. When the real time schedulers do not show positive effects on the performance. We can infer that real time schedulers impact little on the multiple processes which runs on separated cores.

3.2. Virtualization technologies

Virtualization offers multiple execution environments on a single server. Comparing with the monolithic kernel OS, according to Equation 5, the number of guest OS or containers will amplify the outlier, and hence make the outlier proportion deteriorates. We use LXC [34] and Xen[9] as the basic virtualization system to evaluate the outlier proportions of virtualization. The versions of LXC and Xen are 0.7.5 and 4.0.0,

respectively. The VMs on Xen are all para-virtualized. For both LXC and Xen, the node resources are divided equally for the 4 containers and 4 VMs.

LXC is based on a monolithic kernel which binds together multiple processes (process group) to run as a full-functioned OS. A container-based system can spawn several shared, virtualized OS images, each of which is known as a container. It consists of multiple processes bound together (process group), appearing as a full-functioned OS with an exclusive root file system and a safely shared set of system libraries. For Xen-like hardware level virtualization, there is a microkernel-like hypervisor to manage all virtual machines (VMs). Each VM is composed by virtual devices generated by device emulators and runs on a less privileged level. The execution of privileged instructions on a VM must be delivered to hypervisor. Communications are based on the event channel mechanism.

We run memcached instances respectively on four containers and four virtual machines (VMs) hosted on a single node. A request is fanned out to the four containers or VMS and four sub-query responses are merged in the client emulator to calculate the performance. From Figure 3a, we can see that the performance is far from the expectation. Note that when deploying on less than 24 cores, Xen has better outlier proportions. Comparing to LXC, Xen has much better performance isolation. Moreover, Xen's and LXC's outlier proportion significantly deteriorate, respectively when each VM and container runs on 10 cores. We also note that the valid throughput of Xen is much lower than LXC and Linux.

3.3. Discussion

From Figure 3b, we surprisingly find that to reduce the service or job-level outlier proportion to 10%, running Linux or LXC or XEN, respectively, the outlier proportion of a single server needs to be reduced by 871X, 2372X, 2372X accordingly. And for a 5% outlier proportion guarantee, the performance gap is 1790X, 4721X, 4721X, respectively.

Operating system can be abstracted as a multi-thread scheduling system with internal and external interrupts. Outliers are mainly the tasks starved for certain resources because of preemption or interference from parallel tasks or kernel routines. Waiting and serialization can be aggravated by the increasing parallel tasks and cores. Here are a few occasions that the outlier proportion may be degraded.

- **Synchronization.** Synchronization becomes frequency and more time consuming. Such as the lock synchronization of Resource counters, Cache coherence and TLB shutdown between cores. For example, for a multiprocessor OS, TLB consistency is maintained through by invalidating entries when pages are unmapped. Although the TLB invalidation itself is fast, the process of context switches, transferring IPIs (Inter-Processor Interrupts) across all possible cores, and waiting for all acknowledgements may be time consuming [36]. On the one hand, pro-

cessing shutdown IPs need to interrupt current running tasks. On the other hand, the time consumed during transferring and waiting may easily climb rapidly with the increasing cores.

- **Shared status.** Shared status also becomes more common. Such as Shared queues and lists or Shared buffers. For example, information of software resource or kernel states are commonly stored in queues, lists, or trees, such as the per-block list that tracks open files. With increasing tasks on more cores, these structures may be more filled. Thus, searching and traversing them becomes more expensive. Besides, there are many limits set by kernel. In 10 ms computing, the number of tiny tasks may be of large quantity whose accesses of system resources may be easily excessive.
- **Scheduling based on limited hardware.** Hardware resources such as last level cache (LLC), inter-core interconnect, DRAM, I/O hub are physically shared by all processors. Too many loads on the subsystems may reach the capacity up-bounds, so it is difficult for a scheduler to schedule tasks with a optimum solution.

4. Related Work

The outlier problem has been studied in many areas such as parallel iterative convergent algorithms where all executing threads must be synchronized [11]. Within the context of scale-out architecture, we now discuss related work on outlier sources and mitigation.

4.1. Sources of Outliers and Tail Latency

Hadoop MapReduce Outliers. The problem of Hadoop outliers is first proposed in [13] and it is further studied in heterogeneous environments [39]. In Hadoop, the task outliers are typically incurred by task skews, including load imbalance among servers, uneven data distribution, and unexpected long processing time [26, 18, 22, 23, 29]

Tail latency in interactive services. In today’s WCSs, interactive services and batch jobs are typically co-located on the same machine to increase machine utilizations. In such shared environments, resource contention and performance interference is a major source of service time variability [33, 15]. This variability is further significantly amplified when considering requests’ queueing delay, thus incurring high request tail latency.

4.2. Application-level techniques to mitigate outliers

Task/sub-request redundancy is a commonly applied technique to mitigate outliers and tail latency. The key idea of such technique is to execute each individual tasks/sub-request in multiple replicas so as to reduce its latency by using the quickest replica. In [3, 37], this technique has been applied to mitigate outlier tasks in small Hadoop jobs whose number of tasks is smaller than ten. In [35], it is applied to guarantee

low tail latency only when the system at idle state. In contrast, **task/sub-request reissue** is a conservative redundancy technique [4, 19, 29]. This technique first sends a task/sub-request replica to one approximate machine. The replica is judged as the outlier if it is not completed after a short delay (e.g. the 99th percentile latency for this class of tasks/sub-requests), and then a secondary replica is sent to another machine. Both techniques work well when the service is under light load. However, when load becomes heavier, the unnecessarily execution of the same tasks/sub-requests adversely increases the outlier proportion [31].

Partial execution is another widely used technique to mitigate outliers by sacrificing result quality (e.g. prediction accuracy in classification or recommendation services). Following the anytime framework initially proposed in AI [40], this technique has been applied in Bing search engine [19, 16] to return an intermediate and partial search result whenever the allocated processing time expires. Similar approaches have been proposed to sample a subset of input data to produce approximate results for MapReduce jobs under both time and resource constraints [24, 30, 32, 38]. Moreover, best-effort scheduling algorithms have been developed to form a complement to the partial execution technique, which allocate available processing times among tasks/sub-requests to maximize their result quality [17, 16]. However, when load become heavier, such technique incurs considerable loss in result quality to meet response target and this is sometimes unacceptable for users.

5. Conclusion

In this paper we argue computers or smart devices should and will *consistently* provide information and knowledge to human being in the order of a few tens milliseconds despite computation becomes much complex on data with unprecedented large-scale. We coin a new term 10-millisecond computing to call attention to this class of workloads.

We propose a very simple but powerful metric—*outlier proportion* to characterize the system behaviors. As a case study we investigate the challenges raised for conventional operating systems. For a 1K-scale system, a typical internet service configuration, we surprisingly find that to reduce the service-level outlier proportion of a typical latency-critical service—memcached to 10% , running Linux (version 2.6.32) or LXC (version 0.7.5) or XEN (version 4.0.0), respectively, the outlier proportion of a single server needs to be reduced by 871X, 2372X, 2372X accordingly. We also conducted a list of experiments to reveal the current Linux systems still suffer from poor outlier performance, including Linux kernel version 3.17.4, Linux kernel version 2.6.35M, a modified version of 2.6.35 integrated with *sloppy counters* proposed by Boyd-Wickizer et al. and two representative real time schedulers. This observation indicates the new challenges are significantly different from traditional outlier and stagger issues widely investigated in MapReduce and other environments.

References

- [1] Memcached. Accessed Dec 2014. <http://memcached.org/>.
- [2] Mosbench: a set of application benchmarks to measure os scalability. Modified kernels are included. <http://pdos.csail.mit.edu/mosbench/>.
- [3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.
- [4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.
- [5] Michael Barabanov. *A linux-based real-time operating system*. PhD thesis, New Mexico Institute of Mining and Technology, 1997.
- [6] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
- [7] Stuart K Card, Allen Newell, and Thomas P Moran. The psychology of human-computer interaction. 1983.
- [8] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 181–186. ACM, 1991.
- [9] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.
- [10] Byung-Gon Chun and Petros Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS*, volume 9, pages 8–11, 2009.
- [11] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. *HotOS. USENIX Association*, 2013.
- [12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
- [13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [14] Alan Dix. Natural time. In *Position Paper for CHI 96 Basic Research Symposium (April 13-14, 1996, Vancouver, BC)*, 1996.
- [15] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 333–346. USENIX Association, 2013.
- [16] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: scheduling interactive services with partial execution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 12. ACM, 2012.
- [17] Yuxiong He, Sameh Elnikety, and Hongyang Sun. Tians scheduling: Using partial processing in best-effort applications. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 434–445. IEEE, 2011.
- [18] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE, 2010.
- [19] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 219–230. ACM, 2013.
- [20] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 729–742. ACM, 2014.
- [21] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’07, pages 959–967, New York, NY, USA, 2007. ACM.
- [22] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.
- [23] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.
- [24] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.
- [25] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency.
- [26] Jimmy Lin et al. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, volume 1, 2009.
- [27] Marissa Mayer. Rough notes from marissa mayer keynote. keynote at Google IO, 2008.
- [28] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *Proc. HotOS*, 2013.
- [29] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.
- [30] Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11):1135–1145, 2011.
- [31] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. When do redundant requests reduce latency? Technical report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2013.
- [32] Yingjie Shi, Xiaofeng Meng, Fusheng Wang, and Yantao Gan. You can stop early with cola: Online processing of aggregate queries in the cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1223–1232. ACM, 2012.
- [33] David Shue, Michael J Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.
- [34] Stephen Soltész, Herbert Pötzl, Marc E Fluczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.
- [35] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *ICAC*, pages 265–277, 2013.
- [36] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O.S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349, Oct 2011.
- [37] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 13–18. ACM, 2012.
- [38] Yuxiang Wang, Junzhou Luo, Aibo Song, and Fang Dong. A sampling-based hybrid approximate query processing system in the cloud. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 291–300. IEEE, 2014.
- [39] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.
- [40] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.